M257 Unit 11
UNDERGRADUATE COMPUTING

Putting Java to work

Spiders, regular expessions and Google

Unit 11

This publication forms part of an Open University course M257 *Putting Java to work*. Details of this and other Open University courses can be obtained from the Student Registration and Enquiry Service, The Open University, PO Box 197, Milton Keynes MK7 6BJ, United Kingdom: tel. +44 (0)870 333 4340, email general-enquiries@open.ac.uk

Alternatively, you may visit the Open University website at http://www.open.ac.uk where you can learn more about the wide range of courses and packs offered at all levels by The Open University.

To purchase a selection of Open University course materials visit http://www.ouw.co.uk, or contact Open University Worldwide, Michael Young Building, Walton Hall, Milton Keynes MK7 6AA, United Kingdom for a brochure. tel. +44 (0)1908 858785; fax +44 (0)1908 858787; email ouwenq@open.ac.uk

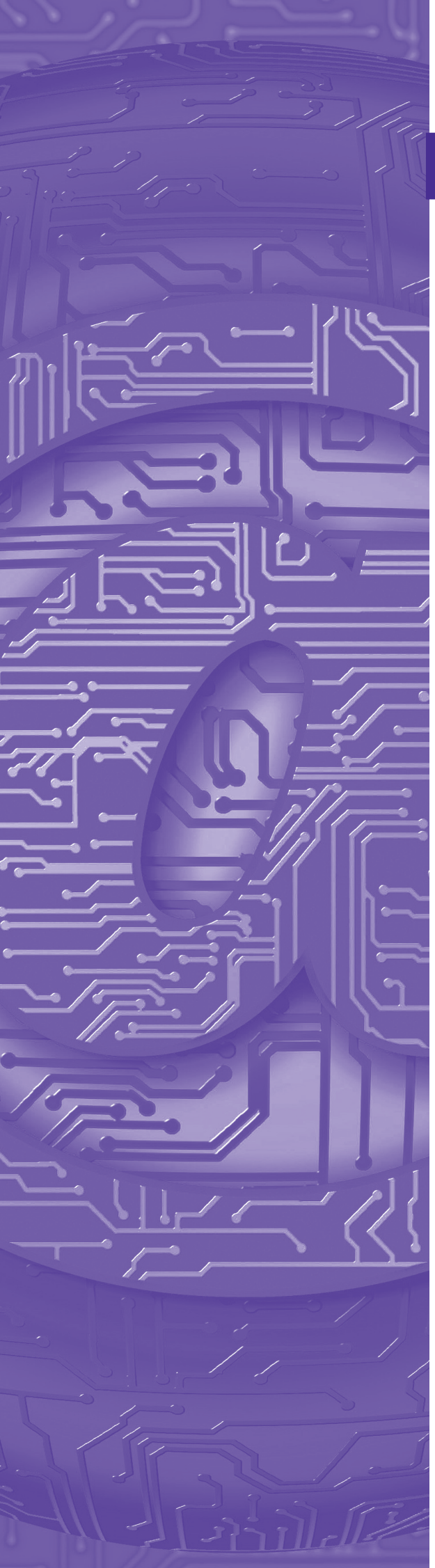
The Open University
Walton Hall, Milton Keynes
MK7 6AA

First published 2007. Second edition 2008.

Edited, designed and typeset for Web publication by The Open University.

WEB 86870 1

2.1



CONTENTS

1 Introduction

Study Note

This unit contains two case studies. You should look at the M257 *Course Guide* and the course website in order to see which case studies you need to study for the current presentation of the course.

Whichever case studies you read you will need to read Sections 1 and 2 of this unit.

Both of the case studies in this unit require you to carry out activities, and both are based on an application area known as text processing in which long strings – usually natural language – are searched. Such searching can be quite complicated and might require some painstaking programming. For example, there could be queries such as:

- Find all the occurrences of the substrings "Northants", "North" and "Northampton" within a web page.
- Find all text "Dear Dave" or "Dear David" within a collection of emails.
- Find all the occurrences of words that have monetary connotations, such as "Dollar", "Dollars", "Pound" and "Pounds".
- Find the number of times that words formed from the verb "dig" appear in the text of an online book.
- Find the number of times a three-letter word occurs next to a four-letter word.

Many of the natural-language processing searches are associated with major applications. For example, the first bullet point above is a typical search that might be carried out by the user of a search engine such as Google. The second bullet point might be a search carried out by a system that manages emails. The third bullet point might be a rule used by a program that detects spam emails: the most common spam emails are aimed at selling you something, so you would expect monetary terms to be prevalent in the emails. The fourth and fifth bullet points might form part of a package to determine who wrote a piece of text. Every writer has a particular style, which is often expressed in concrete terms by metrics that can be collected by relatively simple programs: in the fourth bullet point it is clear that the query is concerned with identifying an author who uses a particular word a large number of times. The fifth bullet point is associated with a search for an author who regularly writes words of a certain length after words of another length. This area of computing is known as literary forensics. It has been used to identify authors of ancient texts and identify criminals, for example blackmailers.

The aims of this unit are to:

- introduce the concept of a regular expression;
- demonstrate how Java can be used to interact with the internet;
- provide you with some practice in reading some moderately long Java programs.

2 Regular expressions

All the queries listed as bullet points in Section 1 can be expressed in terms of Java `if` statements. However, as a query gets more and more complicated the logic of the `if` statements becomes tortuous and hence more error-prone. **Regular expressions** were invented to provide a more compact representation of a string that needs to be searched for. They have been developed in order to reduce the number of `if` statements necessary in a complex search to just one.

Many software systems provide regular expressions for searching; they can be found in Windows Explorer, word processors, many programming languages and in some browsers.

An example of the simplest regular expression that you can specify is:

"The fox saw the duck"

This is a simple string that just matches the substring "The fox saw the duck" within any other string. This regular expression is not hugely useful; it is the regular expression version of zero. A more useful regular expression is

Cat*

The meaning of characters such as * and ? can vary depending on the technology that you are using.

What this regular expression describes is any string consisting of the substring "Ca" followed by zero or more t characters. This means that it identifies strings such as:

Ca
Catt
Catttttt
Catttttttttttttttttttttt

The first string is the simplest case, since it matches "Ca". You may have met a form of regular expressions before, in that when you look for files in a platform such as Windows or Linux you might have used the * character to search for files that contain a number of specific characters.

If you want to match a substring that consists of one or more characters then the + symbol is used:

Cat+

will match any string that starts with "Ca" and is followed by one or more t characters.

You can also specify 'or' options within a regular expression. For example, the regular expression:

A*|b+

will match zero or more capital A characters or one or more b characters.

Brackets can be used to designate substrings. For example, the regular expression:

(ab)*c

matches any string that consists of zero or more occurrences of the substring "ab" followed by the character c. Thus it will, for example, match:

c abc abababc

You can also specify character ranges by using square brackets. For example, the regular expression:

[0-4]*[6-8]*

describes a string that consists of zero or more digits from 0 to 4, followed by zero or more digits from 6 to 8. Thus all the following will be matched:

01 033333333 3332222266887 0000111114444444

Alphabetic characters can also be used within regular expressions in the same way that digits are used. For example, the regular expression:

[a-d]*8*

describes any sequence (including zero) of lower case a, b, c, d followed by zero or more 8 characters; so, for example, the strings:

a aaa88 abcd888888 a8 88 aaaaaaaabbbbbcc abcdabcdabcd

will match the regular expression.

A regular expression can also specify that a particular pattern must be recognized a number of times. This is achieved by means of curly brackets. The regular expression:

B{2}

recognizes two capital letter Bs, while:

B{2,}

matches a string that consists of two or more capital letter Bs. The regular expression B{2,5} matches a string consisting of between 2 and 5 capital Bs.

These are just a few of the facilities that regular expressions offer. Virtually all programming languages offer facilities for specifying regular expressions and applying them to a string in order to discover whether the string consists of particular sub-expression(s) that can be found in the string. Some languages such as Perl, which were defined for string processing, offer extensive facilities for regular expression manipulation, while other languages offer fewer facilities; however, whatever the facility offered, the basic syntax of regular expressions follows the form detailed in this unit.

In Java, regular expression processing is implemented as part of the standard Java libraries. There are also a number of third-party libraries available. The one that I will use in this unit is an API, produced by the Apache Foundation (the Open Source organization that is famous for developing and maintaining the Apache Web server). The API is known as `Regexp`.

Open source software is software whose code can be examined and which is usually free.

`Regexp` is a relatively simple API to use. It is configured around a class `RE`, which defines regular expressions. There is a regular expression library as part of the standard Java distribution but I find the `Regexp` library easier to use.

An example of the type of coding that is involved when using `Regexp` is shown below:

```
RE reg = new RE("x+y+z+");
if(reg.match("abxxyzzzzhello"))
...
```

Here the regular expression object `reg` is created and then the string:

```
"abxxyzzzzhello"
```

is checked against this regular expression.

The string matches the regular expression since it contains a substring `"xxyzzzz"` consisting of one or more `x` characters followed by one or more `y` characters and then one or more `z` characters.

SAQ 1

What will the following regular expression match?

A*b+

ANSWER.....

It will match any number of capital As (including zero) followed by one or more lowercase bs.

SAQ 2

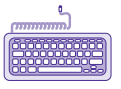
Will the following regular expression match an empty string?

A*b*

ANSWER.....

Yes, since it will match zero or more As followed by zero or more bs.

These, then, are the basic facilities of the `Regexp` API. The two case studies that follow make use of the facilities in the API. Any advanced facilities will be explained when they are met within each case study.



Activity 11.1
Trying out some regular expressions.

In the unit I use the terms 'pattern' and 'pattern matching'. A **pattern** is a string that defines a series of strings; for example, the pattern w* defines the empty string or any sequence of w characters. **Pattern matching** is where a pattern is used to search a string for any substrings that can be described by the pattern.

3 Case study: A spider

One of the most useful internet utilities is known as a spider. A typical spider roams around the World Wide Web examining pages and returning with information about those pages. This case study is concerned with developing a simple spider that visits a specified collection of web pages and determines whether particular content can be found on those pages. It produces a report that details those pages that contain the content.

The particular spider we shall build will examine a database that contains a list of web page addresses and regular expressions, and check whether substrings described by the regular expressions are contained in the pages.

The database that we shall use to contain page details and regular expressions will be defined by the `Properties` class. This is a file-based data structure whose class definition can be found in `java.util`. It is similar in concept to the `HashMap` class that you have met before in the course.

I have chosen to use a `Properties` object because it implements a map with keys and values, which are both strings (this fits our URLs and regular expressions perfectly), and because it is simple to save a `Properties` file and retrieve it later.

The contents of a typical `Properties` file are shown below:

```
David+Robson=HeadOffice
Robert+Key=Accounts
William+Masterton=Marketing
...
```

What we have is the file equivalent of a `HashMap` object. The string to the left of the = symbol represents the key (in the example above this is the name of an employee) and the string to the right of the = symbol is the value associated with the key (in the example above this is the department in which they work). Note that in `Properties` objects the space symbol is represented by the plus sign.

`Properties` objects are normally used to contain the parameters necessary for the running of a program. For example, they might be used to contain information about whether a program starts in beginner, intermediate or advanced mode by having a line such as:

```
Startupmode=advanced
```

within the `Properties` file.

`Properties` objects share many methods with `HashMap` objects. For example, you can send a `get` message to a `Properties` object in order to retrieve the value associated with a particular key.

Within the spider application that we are going to develop, the `Properties` object used will contain the URLs of web pages as the keys and the regular expressions that should be used to search the web pages as the values. For example, the `Properties` object might contain a line such as:

```
http://bbc.co.uk=Fine*last
```

which would indicate that the BBC home page should be searched for a string consisting of "Fin" followed by any number of e characters (including zero) and then the string "last".

The aim of the program that I describe is to process a Properties file with lines such as the one above and then produce a web page that states whether it contains the string described by the regular expression. The report on whether the pattern occurs will be displayed as an HTML file. For example, consider the following HTML file which contains the title "Search report" and a table declaration <TABLE></TABLE>. The table will have a width of 50% of the web page that it is displayed on and a border of one pixel.

```
<HTML>
<HEAD>
<TITLE> Search report</TITLE>
</HEAD>
<BODY>

   <TABLE width="50%" border="1">
      <TR>
         <TD width="72%"><B>URL</B>
         </TD>
         <TD width="28%"><B>Pattern</B>
         </TD>
      </TR>

      <TR>
         <TD width="72%">news.bbc.co.uk/sport1/hi/cricket/counties/
         glamorgan/4254329.stm
         </TD>
         <TD width="28%">Glamorg*n
         </TD>
      </TR>

      <TR>
         <TD width="72%">news.bbc.co.uk/1/hi/wales/mid/4254263.stm
         </TD>
         <TD width="28%">H*y
         </TD>
      </TR>
   </TABLE>

</BODY>
</HTML>
```

Figure 1 shows an example of this table. It contains rows that include the URL of the file that is found, followed by the pattern that is matched. It is worth repeating that such a file will be dynamically generated by the program that I am going to describe in this first case study.

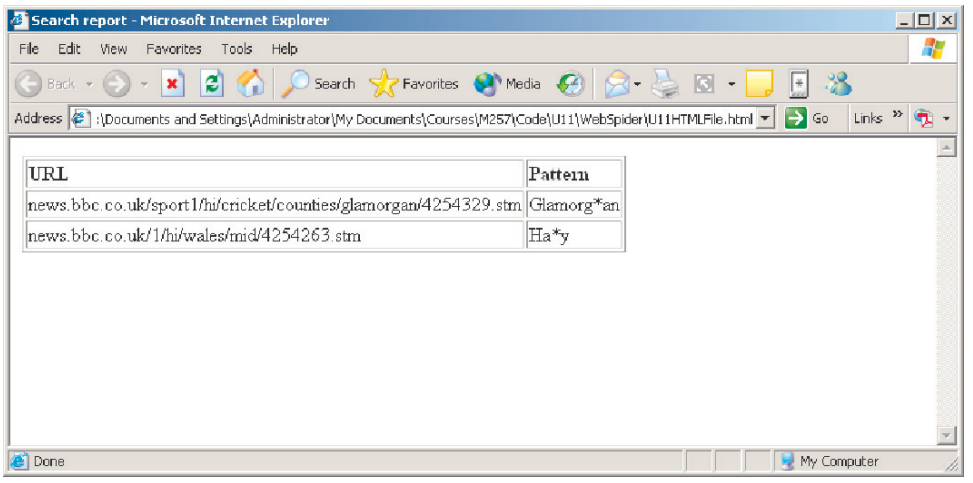


Figure 1 An example of the HTML page produced by the first case study

The file contains a heading for the table, which is described by the following HTML:

```
<TR>
   <TD width="72%"><B>URL</B>
   </TD>
   <TD width="28%"><B>Pattern</B>
   </TD>
</TR>
```

This shows the table as having two columns. The first column contains the word "URL" in bold (delineated by the HTML tags <B>...</B>) and the second column is headed by the word "Pattern", again in bold. The first column will span 72% of the width of the table, while the second column will span 28% of the table.

Following the table heading are two rows of the table. This is specified by the following HTML:

```
<TR>
   <TD width="72%">news.bbc.co.uk/sport1/hi/cricket/counties/
   glamorgan/4254329.stm
   </TD>
   <TD width="28%">Glamorg*
   </TD>
</TR>
<TR>
   <TD width="72%">news.bbc.co.uk/1/hi/wales/mid/4254263.stm
   </TD>
   <TD width="28%">Ha*y
   </TD>
</TR>
```

Here the first row contains in its first column the address of the page:

`news.bbc.co.uk/sport1/hi/cricket/counties/glamorgan/4254329.stm`

and the regular expression pattern that was discovered on it, “Glamorg*an”, while the second row contains the address:

`news.bbc.co.uk/1/hi/wales/mid/4254263.stm`

and the regular expression that was discovered, “Ha*y” (it detected Hay, a small and rather pretty town in east mid-Wales). Note that for simplicity I am assuming that the occurrence of only a single row indicates that one or more occurrences of the regular expression could occur on the web page, although the program will detect only the first.

The skeleton I shall use for the program that processes the `Properties` object and produces the HTML file is shown below:

```
import java.util.*;
import java.io.*;
import java.net.*;
import org.apache.regexp.*;

public class SearchDetails
{
    private Properties propFile;
    private String HTMLFileName;
    ...
}
```

Here a number of `import` statements are used: `java.util` is used for the `Properties` class, `java.io` is used for the various streams needed to communicate with the web pages found in the `Properties` file, `java.net` is used for classes that reference the web page, and `org.apache.regexp` is used to import the Apache regular expression library. The class has two instance variables: the `Properties` object containing the URLs and the regular expressions is `propFile`, and the object that references the HTML file that contains the report is `HTMLFilename`.

The constructor for the class is:

```
public SearchDetails(Properties propFile, String HTMLFileName)
{
    this.propFile=propFile;
    this.HTMLFileName=HTMLFileName;
}
```

All this does is initialize the two instance variables.

The bulk of the processing is carried out within a method called `createReport`. The first task of this method is to create the HTML file and write the header parts of the file. This is shown below:

```
System.out.println("Processing started");
//Create the file to hold the HTML code
PrintWriter HTMLFile=null;
try
{
    HTMLFile =
       new PrintWriter(new FileOutputStream(HTMLFileName),true);
}
catch(Exception e)
{
    System.out.println("HTML file unable to be created");
}
//Write headers to the HTML file
HTMLFile.println("<HTML>");
HTMLFile.println("<HEAD>");
HTMLFile.println("<TITLE> Search report</TITLE>");
HTMLFile.println("</HEAD>");
HTMLFile.println("<BODY>");
```

The code creates a `PrintWriter` object that will clear its print buffer after every `write` statement is executed (the reason for having `true` as the second argument in the creation of the `FileOutputStream` object). In the very unlikely event of the file not being able to be created, an exception is created. After the HTML file is created, a series of HTML statements are written to the file. These statements are just the heading statements for any HTML file.

The next lines establish the variables used for processing the web pages that are to be examined:

```
String uRLValue ="";              //A URL found in the Properties file

String regExps = "";              //A regular expression found
                                  //in the Properties file

URLConnection connect = null;     //Used for establishing
                                  //a connection

InputStream is = null;            //Next two lines used for
                                  //establishing a reader

BufferedReader bReader = null;

String lineRead="";               //Used to hold each line of
                                  //input from a web page

String searchString=null;         //The string that is to be
                                  //searched, i.e. the web page

RE searchPattern = null;          //The search pattern
                                  //(regular expression)

boolean countFound=false;         //Have there been any matches
```

The variables `uRLValue` and `regExps` are used to hold the URL and the regular expression from each line of the `Properties` object. The variable `connect` is used to connect to a web page; later I shall show you how it is used. The variables `is` and `bReader` are used to process a single line at a time from the web page that is currently being examined. The variable `lineRead` is a string used to contain each line of the web page that is being examined. The string variable `searchString` is the string to be searched for that matches to a regular expression; this variable will hold the contents of the web page being examined. The variable `searchPattern` holds the regular expression that is to be used for a search and the `boolean` variable `countFound` is true if a search has been successful.

The next section of processing involves setting up the header of the HTML table that is to be generated; this is shown below:

```
String HTMLtable = "<TABLE width=\"50%\" border=\"1\">\n"+
"<TR>"+
"<TD width=\"72%\"><B>URL</B></TD>"+
"<TD width=\"28%\"><B>Pattern</B></TD>"+
"</TR>\n";
```

Here the string `HTMLTable` is created with the header information in it. Notice the use of the forward slash character to identify double inverted quotation marks, as in `\"50%\"` where the forward slash indicates that the quotation mark after it is to be printed.

The major processing that is carried out within `createReport` is to sequentially examine each line of the `Properties` object, extract out the URL of the web page that is to be examined and then extract out the regular expression that is to be used for searching. A connection to the web page is then established and a string formed from the page. This string is then searched for any substrings that match the regular expression. If a string is found then a line of HTML for the table is formed. The processing completes when the last line of the `Properties` object has been processed.

The code for the processing is shown below:

```
try
{
    for(Object key: propFile.keySet())
    {
        //Get the key and the regular expression
        //for each line in the Properties file
        uRLValue = (String)key;
        regExps = propFile.getProperty(uRLValue);
        //Now get the contents of the page that
        //has been identified
        //First form a URLConnection
        connect = (URLConnection)new
                    URL("http://"+uRLValue).openConnection();
        //Now assign a reader
        is = connect.getInputStream();
        bReader = new BufferedReader
            (new InputStreamReader(is));
        searchString="";
        //Get the contents of the page and place
        //each line in searchString
        lineRead="";
        while(lineRead!=null)
        {
            lineRead = bReader.readLine();
            searchString+=lineRead;
        }
        //Check whether the string is found in the page
        //First create the regular expression that is to
        //be searched for
        searchPattern = new RE(regExps);
        if(searchPattern.match(searchString))
```

Note that on p. 6 of *Unit 9* `URL.openStream` is used to get an input stream. We use as an alternative the method `getInputStream`.

```
        {
            //Search string has been found
            //Issue a line of the table
            HTMLtable+=("<TR><TD width=\"72%\">"+uRLValue
            +"</TD><TD width=\"28%\">"
            +regExps+"</TD></TR>\n");
            countFound=true;
        }
    }
}
catch(Exception e)
{
    System.out.println("Problem setting URL "+e);
}
```

The loop terminates when the `Properties` object has no elements.

The first task in the loop that processes each line of the `Properties` object is to extract out the URL and the regular expression. This is shown below:

```
uRLValue = (String)key;
regExps = propFile.getProperty(uRLValue);
```

Here the first line casts the key and the second line then uses this key to extract out the regular expression associated with the key.

The next stage in the processing is to connect to the web page described by `uRLValue`. This is shown below:

```
connect = (URLConnection)new URL("http://"+uRLValue)
                                    openConnection();
```

This uses the class `URL` and the method `openConnection` to establish a `URLConnection` object that represents a connection to the web page. The method `openConnection` delivers an object described by `Object` and hence casting has to be used to get a `URLConnection`.

The next stage in the processing is to construct a `Reader` object that can read strings from the web page. The `Reader` object is a `BufferedReader`. This allows us to read single lines of text from the page.

```
is = connect.getInputStream();
bReader = new BufferedReader(new InputStreamReader(is));
```

Here the method `getInputStream` gets the input stream associated with the `URLConnection` object `connect` and the `BufferedReader` object `bReader` is then constructed from that stream.

Once a `BufferedReader` object has been formed, the lines making up the web page can be sequentially read. The first part of this processing is shown below:

```
searchString="";
lineRead="";
while(lineRead!=null)
{
      lineRead = bReader.readLine();
      searchString+=lineRead;
}
```

Here each line is read and added to the `String` object `searchString`. The processing finishes when the last line of the web page has been read:

```
(lineRead!=null)
```

The processing continues by forming an `RE` object that represents the regular expression to be searched for and then searching the string that represents the web page. If a string described by the `RE` object is found then a line of the table is added to the string `HTMLTable` and the boolean `countFound` adjusted.

```
searchPattern = new RE(regExps);
if(searchPattern.match(searchString))
{
    HTMLtable+=("<TR><TD width=\"72%\">"+uRLValue
              +"</TD><TD width=\"28%\">"+regExps+"</TD></TR>\n");
    countFound=true;
}
```

The HTML string in the fourth and fifth lines above just writes the value of the URL of the web page and the regular expression associated with that page.

The final processing adds the terminating HTML statements to the HTML file, closes down the HTML file and issues a message that processing has finished. If no patterns were matched the table is not formed and a simple message is placed in the HTML file informing the user.

```
HTMLtable+="</TABLE>";
if(countFound)
      HTMLFile.println(HTMLtable);
else
      HTMLFile.println("No patterns were matched");
HTMLFile.println("</BODY>");
HTMLFile.println("</HTML>");
HTMLFile.close();
System.out.println("Processing finished");
```

The whole code for the class together with comments is shown below:

```
import java.util.*;
import java.io.*;
import java.net.*;
import org.apache.regexp.*;

public class SearchDetails
{
    private Properties propFile;
    private String HTMLFileName;

    public SearchDetails(Properties propFile, String HTMLFileName)
    {
        this.propFile=propFile;
        this.HTMLFileName= HTMLFileName;
    }
```

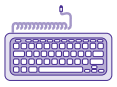
```
public void createReport()
{
    System.out.println("Processing started");
    //Create the file to hold the HTML code
    PrintWriter HTMLFile=null;
    try
    {
        HTMLFile =
            new PrintWriter(new FileOutputStream(HTMLFileName),true);
    }
    catch(Exception e)
    {
        System.out.println("HTML file unable to be created");
    }
    //Write headers to the HTML file
    HTMLFile.println("<HTML>");
    HTMLFile.println("<HEAD>");
    HTMLFile.println("<TITLE> Search report</TITLE>");
    HTMLFile.println("</HEAD>");
    HTMLFile.println("<BODY>");

    String uRLValue ="";                //A URL found in the
                                        //Properties file

    String regExps = "";                //A regular expression found
                                        //in the Properties file

    URLConnection connect = null;       //Used for establishing
                                        //a connection

    InputStream is = null;              //Next two lines used for
                                        //establishing a reader

    BufferedReader bReader = null;

    String lineRead="";                 //Used to hold each line
                                        //of input from a web page

    String searchString=null;           //The string that is to be
                                        //searched, i.e. the web page

    RE searchPattern = null;            //The search pattern
                                        //(regular expression)

    boolean countFound=false;           //Count of the number
                                        //of matches

    //Set up the table header
    String HTMLtable = "<TABLE width=\"50%\" border=\"1\">\n"+
    "<TR>"+
    "<TD width=\"72%\"><B>URL</B></TD>"+
    "<TD width=\"28%\"><B>Pattern</B></TD>"+
    "</TR>\n";
```

```
//Process the properties file line by line.
try
{
    for(Object key: propFile.keySet())
    {
        //Get the key and the list of regular expressions
        //for each line in the Properties file
        uRLValue = (String)key;
        regExps = propFile.getProperty(uRLValue);
        //Now get the contents of the page that has been identified
        //First form a URLConnection
        connect = (URLConnection)new
                    URL("http://"+uRLValue).openConnection();
        //Now assign a reader
        is = connect.getInputStream();
        bReader = new BufferedReader(new InputStreamReader(is));
        searchString="";
        //Get the contents of the page and place
        //each line in searchString
        lineRead="";
        while(lineRead!=null)
        {
            lineRead = bReader.readLine();
            searchString+=lineRead;
        }
        //Check whether the string is found in the page
        //First create the regular expression that is to
        //be searched for
        searchPattern = new RE(regExps);
        System.out.println(uRLValue+" = "+
                              regExps+ "\n"+searchString);
        if(searchPattern.match(searchString))
        {
            //Search string has been found
            //Issue a line of the table
            HTMLtable+=("<TR><TD width=\"72%\">"+
                        uRLValue+"</TD><TD width=\"28%\">"+
                        regExps+"</TD></TR>\n");
            countFound=true;
        }
    }
}
catch(Exception e)
{
    System.out.println("Problem setting URL "+e);
}

//Write footers to HTML file and then close it
HTMLtable+="</TABLE>";
```

```
        //Add table to file if at least one match is found
        //If not send a simple message that no patterns were matched
        if(countFound)
            HTMLFile.println(HTMLtable);
        else
            HTMLFile.println("No patterns were matched");
        HTMLFile.println("</BODY>");
        HTMLFile.println("</HTML>");
        HTMLFile.close();
        System.out.println("Processing finished");
    }
}
```

The only code left to describe is the `main` method that carries out the execution. This is shown below:

```
import java.util.Properties;
import java.io.*;

public class Tester
{
    public static void main(String[] args)
    {
        Properties pageFile = new Properties();
        try
        {
            String pageDetails = ".." ;
            File propsFile = new File(pageDetails);
            pageFile.load(new FileInputStream(propsFile));
            String HTMLOutputFile = "..";
            SearchDetails search = new
                      SearchDetails(pageFile,HTMLOutputFile);
            search.createReport();
        }
        catch(Exception e)
        {
            System.out.println("Properties file not found");
        }
    }
}
```



Activity 11.2
Running and modifying the spider program.

All this does is create the `Properties` object required for the processing by connecting it up to some external file (the name of the file is not shown but is designated by two dots), creating an HTML file (again this is shown by two dots) and then creating the report within the HTML file.

Before leaving the case study program it is worth saying that it is capable of accessing only relatively simple web pages. If, for example, a web page contained frames (a number of different panes on the same window) then the program would not work correctly.

4 Case study: Interfacing to a search engine

This case study deals with writing program code that enables a user to carry out a set of sophisticated searches using a search engine. A **search engine** is a program, accessible via the internet, that enables the user to discover information based on a number of keywords. For example, if you want to find all the web pages that reference Towcester rugby club then you would type in:

```
Towcester Rugby Club
```

There are a number of search engines that are available for free use (at the time of writing in October 2005). Almost certainly the best known and most used is **Google**. This is because it uses software that employs novel methods to return accurate results.

During 2004 the Google developers released a Java API that enabled programmers to develop code that interacts with Google. An example of its use is shown below. Here a simple query is fed to Google and it produces the URL of any web pages that match the query together with the title of the page.

```
import com.google.soap.search.*;

public class GoogleSearchDemo
{
    public static void main(String args[])
    {
        GoogleSearch searchItem = new GoogleSearch();
        searchItem.setKey("...");
        searchItem.setQueryString("\"Towcester Rugby Club\"");
        searchitem.setMaxResults(5);
        // Start the search
        try
        {
            GoogleSearchResult searchResult = searchItem.doSearch();
            // process the results
            GoogleSearchResultElement[] re =
                                searchResult.getResultElements();
            for(int i = 0; i < re.length; i++)
            {
                System.out.println(re[i].getURL() + " " + re[i].getTitle());
            }
        }
        catch(Exception e)
        {
            System.out.println("Problem "+e);
        }
    }
}
```

The word `soap` in the import is an acronym for Simple Object Access Protocol; this is a new web-based protocol that allows user programs to access websites.

The key is not shown here – it has been replaced by three dots.

The import statement:

```
import com.google.soap.search.*;
```

first imports the Java library for the Google API. The line:

```
GoogleSearch searchItem = new GoogleSearch();
```

then creates a `GoogleSearch` object. This is the main object used to retrieve data from the Google search engine's databases. The next line:

```
searchItem.setKey("...");
```

sets a key that has to be used by any program that accesses the Google search engine. Each user of Google is assigned a key when they register for this form of use of the search engine. This key is unique to its user and, at the time of writing (October 2005), allows up to 1000 searches per day.

The next two lines are:

```
searchItem.setQueryString("\"Towcester Rugby Club\"");
searchItem.setMaxResults(5);
```

The first line sets the query that is to be applied to the Google database; notice the escape character \ used to introduce double-quoted characters. The next line specifies that only the top five pages that match the query are to be returned. At the time of writing Google allows you to recover only the top ten pages for a search, although users have been promised that this will change as the Google API becomes more mature. Once these parameters have been set the search can be carried out. This is done via the line:

```
GoogleSearchResult searchResult = searchItem.doSearch();
```

This creates a `GoogleSearchResult` object that represents the result of the search.

The next two lines:

```
GoogleSearchResultElement[] re =
                searchResult.getResultElements();
```

place in the array `re`, which contains `GoogleSearchResultElement` objects, the details of each of the web pages that have been retrieved.

The array can then be iterated through and the details of each page displayed. This is shown below:

```
for (int i = 0; i < re.length; i++)
{
    System.out.println(re[i].getURL()
    + " " + re[i].getTitle());
}
```

Here the URL of the page is displayed together with the title of the page. The title of the page is displayed using HTML commands so that, for example, the first page that was extracted from the query:

```
Towcester Rugby Club
```

is:

```
http://www.towsrus.org/<b>Towcester</b> <b>Rugby</b>
<b>Club</b> Mini and Junior Section
```

where the `<b>...</b>` tags indicate that the text delineated by them will be displayed in bold.

The case study that I will describe here integrates Google searching with the use of the regular expression API detailed in the previous case study. Its functionality is described below:

1 When the program is first invoked a window is shown to the user.

2 The window contains two buttons used for searching. The first will invoke a Google search. The second will invoke a search based on a regular expression. A third button will be used to quit the application.

3 The search terms for the Google search are typed inside a text field. The regular expression searched is typed in a second text field.

4 When the first button is clicked a combo box is populated with the URLs of the pages that have been retrieved from the Google search.

5 When the URLs from the Google search are returned the user can click any one of them in the combo box. This will result in the URL being highlighted. When the regular expression button is then clicked the program will search for any strings matching a typed regular expression. If the page described by the URL contains a string matching the regular expression then this fact is displayed in a third text field; if no match is found then this is announced in the third text field.

6 When the user wishes to exit the program he or she clicks a third button.

An example of the window is shown in Figure 2.

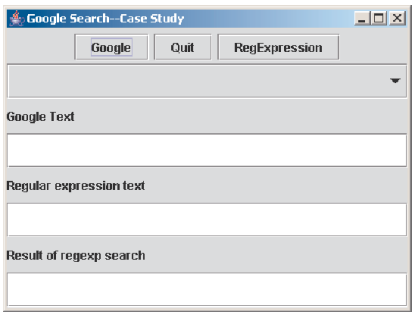


Figure 2 The window created by the second case study

This, then, is an informal specification of the program. It will use facilities detailed here for Google searching, facilities provided by the regular expression library detailed in the previous case study and base Java facilities for creating and displaying visual objects such as buttons and text fields.

The code for this case study is detailed below. It carries out a number of technical functions:

- it creates the visual objects that the user will interact with and lays them out in a window;
- it responds to button-click events;
- when responding to a button click it carries out the function associated with this event.

It is important to point out that there is *no error processing associated with this code* – for example, there is no check that the user has typed in some text for a Google search.

The first code imports all the libraries that are required and declares the visual objects that are to be used:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.google.soap.search.*; //Used for Google search
import java.io.*;
import java.net.*;
import org.apache.regexp.*;

public class QueryWindow extends JFrame
                                    implements ActionListener
{
    JPanel buttonPanel;
    JButton googleButton, regExpButton, quitButton;
    JComboBox URLList;
    JTextField googleText, reText, resultText;
}
```

The three buttons are used to quit the program, carry out a Google search and refine the Google search by using a regular expression. The class `QueryWindow` is a `JFrame` object that listens to action events. A panel is used to hold the three buttons and a combo box is used to hold the URLs that are retrieved when the Google search is invoked. The constructor for the window is shown below:

```
    public QueryWindow()
    {
        //Set up the frame
        super();
        setSize(400, 300);
        setTitle("Google Search-Case Study");
        Container cp = getContentPane();
        cp.setLayout(new GridLayout(8,1));
        //Set up buttons
        buttonPanel = new JPanel();
        googleButton = new JButton("Google");
        regExpButton = new JButton("RegExpression");
        quitButton = new JButton("Quit");
        buttonPanel.add(googleButton);
        buttonPanel.add(quitButton);
        buttonPanel.add(regExpButton);
        //Set remainder of visual objects
        URLList = new JComboBox();
        googleText = new JTextField(30); //Length of 30 chars
        reText = new JTextField(20);     //Length of 20 chars
        resultText = new JTextField(10); //Length of 10 chars
        cp.add(buttonPanel);
        cp.add(URLList);
        cp.add(new JLabel("Google Text"));
        cp.add(googleText);
        cp.add(new JLabel("Regular expression text"));
        cp.add(reText);
        cp.add(new JLabel("Result of regexp search"));
        cp.add(resultText);
```

```
        //Register all listeners
        quitButton.addActionListener(this);
        googleButton.addActionListener(this);
        regExpButton.addActionListener(this);
    }
```

It first sets up a window of length 400 pixels and height 300 pixels, sets the title, and creates a container for the visual objects.

The three buttons are created and added to a panel. Then the combo box and three text fields are created. All the visual objects are then added to the frame, including the three labels that identify the text fields.

Finally, the frame is designated as a listener to all of the three buttons.

The code that is executed when any of the buttons is clicked is shown below. It is part of the `actionPerformed` method, which is implemented in the interface `ActionListener`.

```
    public void actionPerformed(ActionEvent e)
    {
        if(e.getActionCommand().equals("Quit"))
              System.exit(0);
        if(e.getActionCommand().equals("Google"))
        {
            //Remove the result of any previous searches
            URLList.removeAllItems();
            //Now do the Google search
            executeGoogleSearch();
        }
        if(e.getActionCommand().equals("RegExpression"))
            executeReSearch();
    }
```

When the quit button is executed the program exits. When the Google search button is executed the combo box holding the results of the Google search is cleared of any results from a previous search and the method `executeGoogleSearch` is executed. When the regular expression search button is clicked the method `executeReSearch` is executed.

The code for `executeGoogleSearch` is shown below:

```
    public void executeGoogleSearch()
    {
        //Get the search terms used for the Google search
        String searchTerms = googleText.getText();
        //Start a search
        GoogleSearch search = new GoogleSearch();
        //Set the Google search key
        search.setKey("Key goes here");
        //Set the search string and specify
        //that 8 hits are to be returned
        search.setQueryString(searchTerms);
        search.setMaxResults(8);
```

```
        try
        {
            //Carry out the Google search
            GoogleSearchResult searchResult = search.doSearch();
            // process the results
            GoogleSearchResultElement[] re =
                            searchResult.getResultElements();
            //Add the URLs of the Google search results
            //to the JCombo box
            for( int i = 0; i < re.length; i++ )
                        URLList.addItem(re[i] .getURL());
        }
        catch(Exception e)
        {
            System.out.println("Problem "+e);
        }
    }
```

The code first extracts out the terms used for the Google search, creates a new search object, sets the terms for the search and then restricts the number of search items returned to eight.

The code in the `try-catch` statement then executes the search and places each URL that is found in the combo box.

The code for `executeReSearch` is shown below:

```
    public void executeReSearch()
    {
        try
        {
            //Clear result field
            resultText.setText("");
            //Extract out the selected string
            String uRLValue=(String)URLList.getSelectedItem();
            //Get the regular expression to be searched for
            String regExpression = reText.getText();
            //Set up a URL connection
            URLConnection connect = (URLConnection)
                                    new URL(uRLValue).openConnection();
            //Now create a reader
            InputStream is = connect.getInputStream();
            BufferedReader bReader = new BufferedReader
                                     (new InputStreamReader(is));
            String searchString="";
            //Get the contents of the page and place
            //each line in searchString
            String lineRead="";
            while(lineRead!=null)
            {
                lineRead = bReader.readLine();
                searchString+=lineRead;
            }
```

```
            //Check whether the string is found in the page
            //First create the regular expression that is
            //to be searched for
            RE searchPattern = new RE(regExpression);
            //Now check for a match
            if(searchPattern.match(searchString))
               resultText.setText("Found");
            else
               resultText.setText("Not found");
            //Close down all input objects
            bReader.close();
            is.close();
         }
         catch(Exception e)
         {
            System.out.println ("Error in accessing Web page");
         }
      }
```

The code is very similar to that which I described in the previous case study. It first extracts out the URL that the user has selected from the combo box. It then establishes a connection to the web page named by this URL; then a reader is created based on this connection, and the lines of the web page are read and concatenated into the string `searchString`. This string is then searched for any substrings that match the regular expression found in the text field `reText`. If there is at least one match then the text field `resultText` is set to contain the string `"Found"`; otherwise it is set to contain the string `"Not found"`.

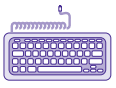
The full code for the case study is shown below:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.google.soap.search.*; //Used for Google search
import java.io.*;
import java.net.*;
import org.apache.regexp.*;

public class QueryWindow extends JFrame
                                 implements ActionListener
{
   JPanel buttonPanel;
   JButton googleButton, regExpButton, quitButton;
   JComboBox URLList;
   JTextField googleText, reText, resultText;

   public QueryWindow()
```

```
{
    //Set up the frame
    super();
    setSize(400, 300);
    setTitle("Google Search-Case Study");
    Container cp = getContentPane();
    cp.setLayout(new GridLayout(8,1));
    //Set up buttons
    buttonPanel = new JPanel();
    googleButton = new JButton("Google");
    regExpButton = new JButton("RegExpression");
    quitButton = new JButton("Quit");
    buttonPanel.add(googleButton);
    buttonPanel.add(quitButton);
    buttonPanel.add(regExpButton);
    //Set remainder of visual objects
    URLList = new JComboBox();
    googleText = new JTextField(30); //Size of 30 chars
    reText = new JTextField(20); //Size of 20 chars
    resultText = new JTextField(10); //Size of 10 chars
    cp.add(buttonPanel);
    cp.add(URLList);
    cp.add(new JLabel("Google Text"));
    cp.add(googleText);
    cp.add(new JLabel("Regular expression text"));
    cp.add(reText);
    cp.add(new JLabel("Result of regexp search"));
    cp.add(resultText);
    //Register all listeners
    quitButton.addActionListener(this);
    googleButton.addActionListener(this);
    regExpButton.addActionListener(this);
}

public void actionPerformed(ActionEvent e)
{
    if(e.getActionCommand().equals("Quit"))
              System.exit(0);
    if(e.getActionCommand().equals("Google"))
    {
        //Remove the result of any previous searches
        URLList.removeAllItems();
        //Now do the Google search
        executeGoogleSearch();
    }
    if(e.getActionCommand().equals("RegExpression"))
        executeReSearch();
}
```

```
public void executeGoogleSearch()
{
    //Get the search terms used for the Google search
    String searchTerms = googleText.getText();
    //Start a search
    GoogleSearch search = new GoogleSearch();
    //Set the Google search key
    search.setKey("Key goes here");
    //Set the search string and specify
    //that 8 hits are to be returned
    search.setQueryString(searchTerms);
    search.setMaxResults(8);
    try
    {
        //Carry out the Google search
        GoogleSearchResult searchResult =
            search.doSearch();
        // process the results
        GoogleSearchResultElement[] re =
            searchResult.getResultElements();
        //Add the URLs of the Google search results
        //to the JCombo box
        for( int i = 0; i < re.length; i++ )
            URLList.addItem(re[i].getURL());
    }
    catch(Exception e)
    {
        System.out.println("Problem "+e);
    }
}

public void executeReSearch()
{
    try
    {
        //Clear result field
        resultText.setText("");
        //Extract out the selected string
        String uRLValue=(String)URLList.getSelectedItem();
        //Get the regular expression to be searched for
        String regExpression = reText.getText();
        //Set up a URL connection
        URLConnection connect = (URLConnection)
                                    new URL(uRLValue).openConnection();
        //Now create a reader
        InputStream is = connect.getInputStream();
        BufferedReader bReader = new BufferedReader
                                    (new InputStreamReader(is));
        String searchString="";
        //Get the contents of the page and place
        //each line in searchString
        String lineRead="";
        while(lineRead!=null)
```

```
            {
                lineRead = bReader.readLine();
                searchString+=lineRead;
            }
            //Check whether the string is found in the page
            //First create the regular expression that is
            //to be searched for
            RE searchPattern = new RE(regExpression);
            //Now check for a match
            if(searchPattern.match(searchString))
                  resultText.setText("Found");
            else
                  resultText.setText("Not found");
            //Close down all input objects
            bReader.close();
            is.close();
        }
        catch(Exception e)
        {
            System.out.println("Error in accessing Web page");
        }
    }
}
```



Activity 11.3
Running and modifying the search program.

The class can be simply executed via the class `Tester` shown below:

```
public class Tester
{
    class Tester
    {
        public static void main(String[] args)
        {
              new QueryWindow().setVisible(true);
        }
    }
}
```

5 Summary

In this unit you have looked at two case studies that carry out internet-based processing. Both of the case studies rely on a concept known as a regular expression. The first case study searched web pages for regular expressions and the second case study provided a sophisticated regular-expression-based search facility that employed the Google search engine.

LEARNING OUTCOMES

When you have completed this unit, you should be able to:

- develop small programs that employ regular expressions;
- modify the code of the first case study to enhance its functionality;
- modify the code of the second case study to change its functionality;
- understand Java programs that use regular expressions;
- understand Java programs that use the Google API.

Concepts

The following concepts have been introduced in this unit:

Google, pattern, pattern matching, regular expression, search engine.

Index